Homework 11
APPM/MATH 4650 Fall '20 Numerical Analysis

Due date: Saturday, December 5, before midnight, via Gradescope. **Instructor**: Prof. Becker
Theme: Absolute stability, Gaussian elimination

Instructions Collaboration with your fellow students is OK and in fact recommended, although direct copying is not allowed. The internet is allowed for basic tasks (e.g., looking up definitions on wikipedia) but it is not permissible to search for proofs or to *post* requests for help on forums such as `http://math.stackexchange.com/` or to look at solution manuals. Please write down the names of the students that you worked with.

An arbitrary subset of these questions will be graded.

Turn in a PDF (either scanned handwritten work, or typed, or a combination of both) to **Gradescope**, using the link to Gradescope from our Canvas page. Gradescope recommends a few apps for scanning from your phone; see the Gradescope HW submission guide.

We will primarily grade your written work, and computer source code is *not* necessary except for when we *explicitly* ask for it (and you can use any language you want). If not specifically requested as part of a problem, you may include it at the end of your homework if you wish (sometimes the graders might look at it, but not always; it will be a bit easier to give partial credit if you include your code).

Problem 1: Absolute stability of RK4. Recall that the RK4 method is defined by

$$\begin{aligned} k_1 &= hf(t_i, w_i) \\ k_2 &= hf(t_i + h/2, w_i + k_1/2) \\ k_3 &= hf(t_i + h/2, w_i + k_2/2) \\ k_4 &= hf(t_i + h, w_i + k_3) \\ w_{i+1} &= w_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

a) Show that the polynomial $Q(h\lambda)$ (equation 5.66 in Burden and Faires) for RK4 is

$$Q(h\lambda) = 1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4,$$

i.e., that $w_{i+1} = Q(h\lambda)w_i$ when applied to the test equation $y' = \lambda y$.

b) Consider the ODE $y'' + 4y' + 13y = 13\cos(t/10)$. Find the homogeneous solution to this ODE (this should be pen-and-paper, no programming involved).

c) We're interested in solving the IVP for the above ODE with $y(0) = 1$, $y'(0) = 0$ on the interval $t \in [0, 100]$. Suppose we solve this using RK4 with a stepsize of $h = 3/4$. What can you say about the solution? Explain your answer (again, no programming needs to be involved).

Problem 2: Gaussian elimination

a) Implement your own code (in Matlab or Python) to perform Gaussian elimination and then back-substitution to solve a square system of equations $A\mathbf{x} = \mathbf{b}$.[1] Include your code in your homework. For simplicity, you do *not* have to do any kind of pivoting (we'll optimistically assume that we never encounter a zero pivot).

[1] For Python users, be aware that `numpy` distinguishes between a pure vector of length n (which has "shape" `(n,)`) and a row vector of length n (shape `(1,n)`) and a column vector of length n (shape `(n,1)`). Often a pure vector is treated like a column vector, but not always. You can convert between the different shapes using things like `numpy.reshape`, `numpy.atleast_2d`, and `.T` (to transpose things). Other helpful `numpy` things may include `numpy.hstack` or `numpy.concatenate`, and `@` for matrix multiplication.

b) Let $n = 10$ and define the $n \times n$ matrix $A = [A_{ij}]$ as

$$A_{ij} = \cos\left(\frac{(i-1)(j-\frac{1}{2})\pi}{n}\right) \quad \forall i, j = 1, \ldots, n$$

so it should look like

```
1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
0.9877    0.8910    0.7071    0.4540    0.1564   -0.1564   -0.4540   -0.7071   -0.8910   -0.9877
0.9511    0.5878    0.0000   -0.5878   -0.9511   -0.9511   -0.5878   -0.0000    0.5878    0.9511
0.8910    0.1564   -0.7071   -0.9877   -0.4540    0.4540    0.9877    0.7071   -0.1564   -0.8910
0.8090   -0.3090   -1.0000   -0.3090    0.8090    0.8090   -0.3090   -1.0000   -0.3090    0.8090
0.7071   -0.7071   -0.7071    0.7071    0.7071   -0.7071   -0.7071    0.7071    0.7071   -0.7071
0.5878   -0.9511   -0.0000    0.9511   -0.5878   -0.5878    0.9511    0.0000   -0.9511    0.5878
0.4540   -0.9877    0.7071    0.1564   -0.8910    0.8910   -0.1564   -0.7071    0.9877   -0.4540
0.3090   -0.8090    1.0000   -0.8090    0.3090    0.3090   -0.8090    1.0000   -0.8090    0.3090
0.1564   -0.4540    0.7071   -0.8910    0.9877   -0.9877    0.8910   -0.7071    0.4540   -0.1564
```

This is a version of the Discrete Cosine Transform (the DCT-II). Let $\mathbf{x} = (1, 2, \ldots, n)^T$, and define $\mathbf{b} = A\mathbf{x}$.

Apply your Gaussian elimination code to solve $\mathbf{b} = A\mathbf{x}$ for $\mathbf{x}$, and report the augmented matrix (with 2 decimal places) after it has become upper triangular, and also the entrywise error in your final solution for $\mathbf{x}$.

c) Let $n = 1000$ and A be a $n \times n$ matrix with entries from the standard normal distribution[2], and again define $\mathbf{x} = (1, 2, \ldots, n)^T$ and define $\mathbf{b} = A\mathbf{x}$. How long does it take your Gaussian elimination code to solve this system? And what's the error (in terms of Euclidean norm, `norm` in Matlab and `numpy.linalg.norm` in Python)?

Hint: for timings, in Matlab you can use `tic` and `toc`; in Python, you can import the `time` module and use `time.perf_counter()`, or

```
from timeit import time
%time [your command here]
```

which is an IPython line magic (so works in IPython and Jupyter). Fun fact: IPython was created as a side project by CU Boulder student Fernando Pérez in 2001.

d) Solve the same system of equation as in (c) but use Matlab's/Python's default solvers, and report the time. For Matlab, this is "backslash" `A\b` aka `mldivide`, and in Python, this is `scipy.linalg.solve`. Also report the error, and comment on the differences between (d) and (c).

Problem 3: LU factorization verse the inverse We're interested in solving the square $n \times n$ system of equations $A\mathbf{x} = \mathbf{b}$ using standard techniques; we'll do some programming, but it will be high-level programming. To recall LU-factorization, we write

$$A\mathbf{x} = \mathbf{b}, \quad A = \underbrace{P \cdot L \cdot U}_{\text{pivoted LU factorization}}, \quad \text{so} \quad P \cdot L \cdot \underbrace{U \cdot \mathbf{x}}_{\mathbf{y}} = \mathbf{b}$$

where P is a permutation matrix, either given as an explicit matrix (and it is orthogonal, so $P^T = P^{-1}$ hence $LU\mathbf{x} = P^T\mathbf{b}$) or as a permutation vector `p` of the indices so we can apply $P^T b$ as `b(p)` (Matlab) or `b[p]` (Python). Then we first solve $L\mathbf{y} = P^T b$ using *forward-substitution*, and then solve $U\mathbf{x} = \mathbf{y}$ using *backward-substitution*. Be careful: Matlab and Python have different conventions for returning P — Matlab actually returns P^{-1} while Python returns P.

a) Write high-level code that solves $A\mathbf{x} = \mathbf{b}$ using these three methods:

i. (Highest level) Use "backslash" `A\b` aka `mldivide` (Matlab) or `scipy.linalg.solve` (Python)

[2] in Matlab, `randn(n)`; and in Python, `from numpy.random import default_rng` then `rng = default_rng()` then `A = rng.standard_normal((n,n))`

ii. (LU with pivoting) Perform an LU factorization and then solve via backsubstitution, relying on Matlab or Python libraries for everything. In particular, for Matlab, use either `[L,U,P] = lu(A)` or `[L,U,p] = lu(A,'vector')` (slightly faster), and figure out how to use the permutation matrix; then you can call something like `U\y` and Matlab will know that `U` is upper triangular and use backsubstitution. For Python, use either `scipy.linalg.lu(A,permute_l=False)` followed by some `scipy.linalg.solve_triangular` (for the forward/backward substitution; sometimes with the flag `lower=True`) and applying the inverse permutation to b; or use `scipy.linalg.lu_factor(A)` followed by `scipy.linalg.lu_solve`. Note that neither Matlab nor `scipy` has an LU factorization function that doesn't pivot; if it doesn't appear to pivot, it means that it really has pivoted but just incorporated the permutation into L.

iii. (Inverse) Solve via an explicit inverse of A, using `inv` (Matlab) or `scipy.linalg.inv` (Python).

Show your code (each method might only be one or two lines of code). It's a good idea to check your code on a small problem with a known answer to make sure all these methods work.

b) Record the timing and error for each of these methods for a range of sizes n between 1000 and 5000, and plot both the error and the timing; is the timing $O(n^2)$ or $O(n^3)$ or $O(n^4)$? For a given n, generate $A, \mathbf{x}$ and $\mathbf{b}$ as we did in Problem 2c/d. Comment on your findings.

Hint: for timings, in Matlab you can use `tic` and `toc`; in Python, you can import the `time` module and use `time.perf_counter()`.